JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 311–337 (1999)

Research

VSAM Conversion for COBOL Programs†

HAI HUANG[1]* and WEI-TEK TSAI[2]

[1]*Guidant Corporation, 4100 Hamline North Avenue, Saint Paul MN 55112-5798, USA*
[2]*Department of Computer Science and Engineering, University of Minnesota, 200 Union Street, 4–192, EE/CS Building, Minneapolis MN 55455-0154, USA*

SUMMARY

VSAM database systems are widely used on IBM mainframe systems. COBOL is the most frequently used host language to access VSAM databases. To fix the Year 2000 bug and migrate to new client/server technology, business organizations are re-engineering their legacy COBOL programs and converting VSAM databases to SQL databases. This paper addresses two important problems in VSAM conversion: database schema conversion and SQL query generation. It proposes a semi-automatic approach to the conversion of VSAM data sets to SQL tables and the conversion of VSAM operations to embedded SQL statements. The proposed approach and a prototype tool have been tested on COBOL programs from industry. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: code migration; SQL; database access methods; code restructuring; schema conversion; query generation

1. INTRODUCTION

The virtual storage access method (VSAM) is the most popular data set organization and access method on IBM mainframe systems. According to Brumbaugh (1993, p. xv), approximately 50% of all business application programming is done on IBM compatible mainframe computers, 95% of which use VSAM on at least some applications. Application programs can use a variety of constructs in high-level languages like COBOL or PL/1, to create and access VSAM data sets. COBOL is the most frequently used language. As new technologies such as relational database and client–server computing become popular, their advantages become visible as:

- Modern relational database systems are Y2K compliant.
- Relational database systems support multiple platforms with open architecture.
- Standardized interfaces simplify access via high-level query languages like SQL.
- Client–server systems have better scalability than monolithic mainframe systems.

Such advantages lead to the migration from monolithic mainframe systems to open client–server systems. This trend requires a re-engineering of the existing legacy systems (Ong and Tsai, 1993; Rugaber and Doddapaneni, 1993; Chen *et al.*, 1994; Gray, Bickmore and Williams, 1995; Huang

*Correspondence to: Dr. Hai Huang, Guidant Corporation, 4100 Hamline North Avenue, Saint Paul MN 55112-5798, USA. Email: hai.huang@guidant.com

†The opinions expressed in this paper reflect the authors' personal opinions only, and do not relate to Guidant's activities.

CCC 1040–550X/99/050311–27$17.50

Received 16 July 1998
Revised 28 July 1999

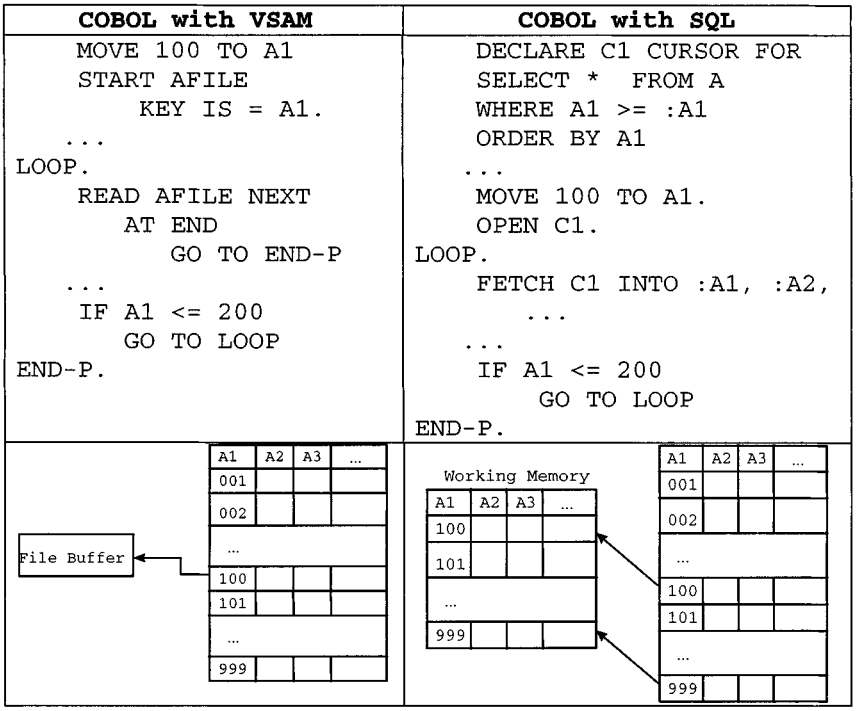

Figure 1. Example of a simple-minded code conversion from using VSAM to using SQL

et al., 1998; Polak, Nelson and Bickmore, 1995). Conversion of VSAM access methods to SQL will benefit the migration progress.

Database re-engineering involves schema conversion, data conversion and application code conversion. One of the major issues in application code conversion is performance degradation, which is often due to inefficient translation of VSAM operations to SQL queries. For example, consider the COBOL code listed in the first column of Figure 1. It reads all the records that have key values between 100 and 200 from the file `AFILE`. Since the records have unique key values and are sorted in ascending order, only 100 records are read. However, if it were converted to the code listed in the second column, once `OPEN C1` is executed, all the records after record 100 are read into the working memory with 900 records in total. This significantly degrades the performance.

The organization of this paper is as follows. Section 2 overviews our approach to making a VSAM to SQL migration. Section 3 deals with schema conversion, and Section 4 with the identification of VSAM input–output operations. Section 5 considers VSAM operation conversion. Sections 6, 7, and 8 briefly cover the restructuring of iterative loops, implementation matters, and verification and validation. The conclusions are in Section 9.

2. STEPS OF THE APPROACH

This section presents an overview of a data-centered *semi-automatic* approach which assists

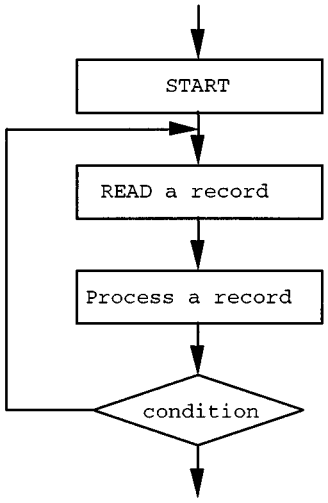

Figure 2. Sequential processing using VSAM

software maintainers in performing the conversion of database schema and the COBOL application code. The approach has the following four steps:

- Identify VSAM data set schema and convert them to SQL relations.
- Identify input-output (I/O) operations that manipulate the VSAM data sets.
- Generate functionally equivalent SQL statements.
- Restructure the code.

In the first step, we convert the VSAM data set schema to SQL relations. Data set definitions include property and record description. Although parsing the source program can capture these data set definitions, it is not a trivial job to convert them to SQL tables. Section 3 discusses this step in detail.

In the second step, we identify I/O operations that manipulate the VSAM data sets. Eight types of statement are available in a COBOL program to perform operations on VSAM data sets. Since each I/O statement always specifies the logical ID (or record name) of a data set it accesses, the data sets can be identified by automatic parsing techniques. Section 4 examines this step in detail.

In the third step, we generate functionally equivalent SQL statements to replace VSAM operations. In COBOL with VSAM applications, sequential processing is most frequently used, and processes a sequence of consecutive records in a data set (Figure 2). Using embedded SQL, a host COBOL program also uses a similar loop structure (Figure 3) to process multiple rows in a table, because COBOL does not have any single operation that simultaneously manipulates a set of records. A comparison of Figures 2 and 3 suggests that a VSAM sequential processing loop can be replaced with an embedded SQL processing loop. The most difficult part of this task is to generate a SQL cursor declaration with the query expression retrieving exactly the same records as in the VSAM loop. Section 5 investigates this step in detail.

In the last step, we restructure the code by adding the SQL statements to and removing the VSAM operations from the COBOL code. Simple deletion of the VSAM components may destroy the

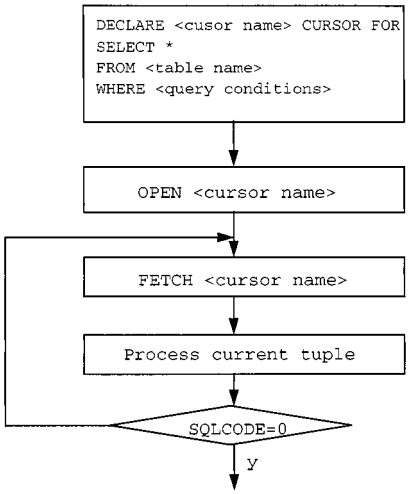

Figure 3. Sequential processing using SQL

correct control sequence of the original COBOL program. For example, an IF statement controlling the execution of a VSAM I/O operation may also control the execution of other COBOL statements, and hence the IF statement cannot simply be removed. Similarly, the positions to insert the SQL statements cannot be arbitrary, because the host variables used in the SQL query must obtain correct values before the query can be evaluated. Section 5 includes a discussion of this step.

3. SCHEMA CONVERSION

3.1. Types of VSAM data set

There are three types of VSAM data set:

- KSDS is an indexed data set, and a unique key is used to locate the record randomly.
- ESDS is a non-indexed data set and records are ordinarily processed in the same order in which they were written to the data set.
- RRDS is a non-indexed data set, and records are stored in fixed length slots and identified by a relative record number that indicates their relative positions in the file.

This paper covers only KSDS and ESDS data sets. We do not have an automatic approach to convert RRDS data sets. Fortunately, RRDS is less frequently used than KSDS or ESDS.

3.2. Identification of data set definitions

Data set definitions include property and record descriptions. A `SELECT` statement in the `ENVIRONMENT` division of a COBOL program specifies the properties of a data set. A 01 level

statement associated with an FD statement defines the record structure of the data set. Therefore, we can capture the property and record structure information by parsing the SELECT and 01 statements following the relevant FD statements in a COBOL program.

More specifically, a SELECT defines the type (i.e., KSDS, RRDS, or ESDS), the keys (for KSDS or RRDS), access mode, and other properties of a data set. Briefly summarized without all the format options, its syntax is:

```
SELECT <file name>
ASSIGN TO <DD name>
ORGANIZATION IS <INDEXED | SEQUENTIAL | RELATIVE>
ACCESS IS <RANDOM | SEQUENTIAL | DYNAMIC>
RECORD KEY IS <variable name>
FILE STATUS IS <variable name>
{other clauses}
```

The <file name> specifies the name of the data set and connects to an FD statement in the FILE section of the DATA division. I/O operations on the data set in the PROCEDURE division may refer to this <file name> too. In addition to the <file name>, we are also interested in the following clauses in the SELECT statement:

- An ACCESS clause specifies one of three possible access modes—SEQUENTIAL, RANDOM, and DYNAMIC. The different access modes determine the different semantics of an I/O operation on the data set, which require different translation schemes to SQL statements.

Example. The semantics of a READ statement are different for each access mode. A random READ statement gets the record that has a key equal to the value stored in the variable specified in the RECORD KEY clause. A sequential READ statement gets the record that the file cursor currently points to, and advances the file cursor to the next record.

- A RECORD KEY clause is used only for KSDS, which designates a field of the record as the primary key of the data set. In the RANDOM access mode, the key is particularly important because it determines the record to be accessed in the data set.
- A FILE STATUS clause defines the field where a status code is placed following every I/O operation on the data set. The status code identifies whether an operation is successfully performed or not. If the status code is non-zero, an error has occurred in the operation.

An FD statement in the FILE section is always followed by a 01 statement. The 01 statement defines the record structure of the data set specified in the FD statement. The format of an FD statement is shown below:

```
FD <file name> {other clauses}
```

Figure 4 and Table 1 illustrate the relations between a SELECT statement, an FD statement, and a 01 statement. Thus, by parsing the SELECT statements in the FILE section, we can capture the schema and important properties of the data sets for the use of later conversion.

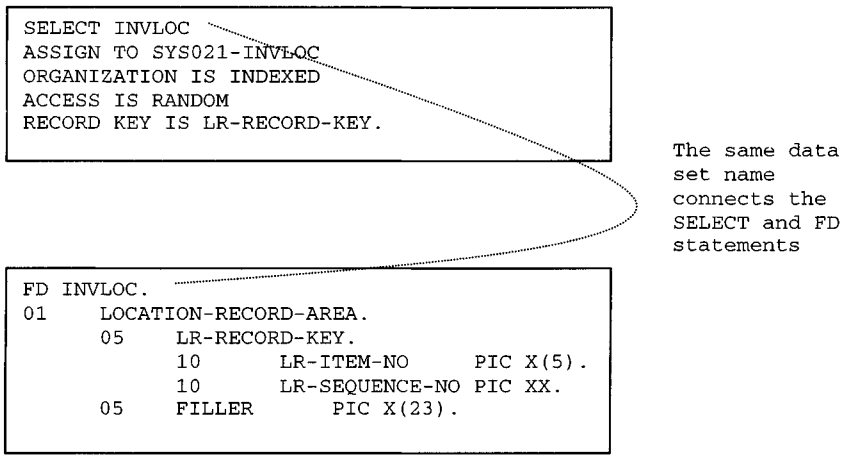

Figure 4. The connection between a SELECT *and an* FD *statement*

Table 1. Data set information of interest

Data set name	Schema (Record description)	Organization (Type)	Access mode	Keys
INVLOC	LOCATION-RECORD-AREA	KSDS INDEXED	RANDOM	LR-RECORD-KEY

3.3. Conversion of VSAM data sets to SQL relations

In a similar way to relational database design, network and hierarchical database design also needs to deal with data redundancy and potential inconsistency problems. Techniques such as virtual fields and virtual record types are used to solve these problems (Ullman, 1988). In this paper, we assume the network databases or the hierarchical databases have been properly designed to avoid data redundancy. When converting logical record types to relations, we simply create one relation for one record type and convert each field in the record format to an attribute of the relation. The resulting relations are at least in the first normal form (1NF). We do not attempt to normalize the resulting relations to third normal form (3NF) if they are not after the simple conversion described above. Intuitively, this conversion approach should not increase the burden of maintaining the data consistency in the migrated relational database, because the resulting relations have at least the same degrees of normalization as their counterparts in the network or hierarchical model.

Another reason for taking this simple conversion approach is to keep the logical view of the database unchanged. With this approach, the control flow and data flow of the COBOL application programs only need small changes, and hence, the functional equivalence is easier to preserve and verify. If we perform additional normalization, the resulting relation schemes could be different from the original record formats. This would require significant changes to the control flow and/or

data flow of the COBOL application programs that access the database, and makes automated conversion of the application programs harder. One may argue that we can normalize the relations and create views that are consistent with the original logical record formats. This is feasible only if the application programs do not need to update the database. Currently, no relational database systems support complex view update. Finally, using this conversion approach, it will be easier to update the documentation of database schemas as there is one-to-one correspondence between the resulting relation and the original logical record format.

The following steps summarize our proposed approach:

- Flatten the VSAM record formats, eliminating fields at the intermediate levels and renaming fields if there are naming conflicts for the lowest-level fields.
- Remove the file control sections, i.e., `SELECT` statements and `FD` statements.
- Add a host variable declaration section to the `DATA` division, since every attribute of an SQL table has a corresponding host variable, and embedded SQL uses host variables to exchange data between the tables and a host program.

COBOL records usually have multiple levels. It is not a trivial task to flatten the record. The following two issues need to be addressed:

- The field definitions of a single record type may be spread across multiple source files. Each file may define only the relevant fields of a data set and use `FILLER` fields for the remaining fields. The complete record structures can be obtained by parsing all the related source files.
- A VSAM data set may contain different types of records; namely, a data set `FD` statement can have multiple record descriptions. Even worse, the `REDEFINES` clause can redefine a previous structure to a different one (similar to a union in the C language).

For the first case, we create one record variable in the `WORKING STORAGE` section for each record description of the data set. Only one of the multiple record descriptions is flattened into a relational database table. Every time a row is retrieved from the table, all the attributes are assigned to a set of host variables. This set of host variables is then used to assign values to the fields of the working storage record, the structure of which is identical to the one flattened. Finally, this working storage record is used to assign values to the other record variables corresponding to other record descriptions.

For the second case, redefining fields in the `REDEFINES` clause are not included in the flattened table. Only one host variable is introduced for the overlapped area. Since we have introduced a working storage record variable for each record description, the redefining fields will automatically get values when the overlapped area is defined.

4. IDENTIFICATION OF VSAM I/O OPERATIONS

Nine types of I/O statements are available in a COBOL program to perform I/O operations on VSAM data sets (Brumbaugh, 1993, p. 237). They are:

- `OPEN`—Prepare the data set for processing (used before the first I/O operation on it).
- `CLOSE`—Terminate accessibility of the data set (used after the last I/O operation on it).
- `READ NEXT`—Specify a sequential read operation with a KSDS or an RRDS.

- `READ`—Specify a random read operation using the current key of reference with a KSDS and slot number with an RRDS. With ESDS, this results in a sequential read operation.
- `READ KEY`—Specify a KSDS read operation of the key.
- `WRITE`—Insert a record into the data set.
- `REWRITE`—Replace an existing record in a data set. It can be a random or sequential operation.
- `DELETE`—Physically remove one record from a KSDS or RRDS organized data set.
- `START`—Establish a position in the data set for resuming the processing of records.

The I/O operations on each data set can be automatically identified because every I/O statement in COBOL must specify the data set it operates on.

5. VSAM OPERATION CONVERSION

5.1. Access modes

A data set can be accessed in one of the following three access modes:

- **Random access** mode allows access to records in a data set in any order (not applicable to ESDS data sets because usually a record key field must be used).
- **Sequential access** mode allows access to logically continuous records, one after the other, in the order in which they are currently in the data set.
- **Dynamic access** mode is a mixture of sequential and random access modes (not applicable to ESDS).

Different access modes require different conversion strategies. In Section 5.2, we shall focus on the conversion of the sequential access mode. Then in Section 5.3, we shall discuss the conversion of the random access mode, and the issues for the conversion of the dynamic access mode.

5.2. Sequential operations

5.2.1. Logical basis

In COBOL applications, sequential processing is often used to process a sequence of consecutive records in a data set. The processing begins with a record that is located by a START statement or the first record by default, and continues until stopped, or until the end of the data set (end of file) is reached. If the data set has been ordered by sorting, then the records are accessed in an ascending (or descending) order, as determined by the prior sorting on a 'key' (data within each record). In KSDS, the records are sequentially accessed in ascending order of the file's key. In ESDS, the records are sequentially accessed in the physical order in the file. Figure 2 illustrates a sequential processing model commonly used in COBOL applications. Except for the different semantics of the record order between KSDS and ESDS, the coding and logic for accessing an ESDS is almost the same as for a KSDS.

Similarly, using embedded SQL, the host COBOL program must also use a loop to process multiple rows, because many imperative languages like COBOL do not have any single command that manipulates a set of records simultaneously. In practice, we usually use SQL cursor declaration together with a COBOL loop construct to perform the task. Figure 3 illustrates this processing model. Note that the rows retrieved by an SQL query are not necessarily consecutive in the original table. Thus, an SQL query combined with a COBOL loop construct is more powerful than a VSAM processing loop.

Furthermore, there is an important difference between Figures 2 and 3. VSAM retrieves and processes records one by one, starting from the first record and continuing to the last one in the sequence. In contrast, embedded SQL retrieves all the records to be processed from the table by a single SQL query and then processes them one by one.

Based upon a recognition of the similarities between such logic, as shown in Figures 2 and 3, we convert VSAM sequential operations by replacing the VSAM loop with an embedded SQL loop. This replacement process has three steps:

- Generate SQL cursor declarations.
- Replace `READ` and `WRITE` statements with SQL `FETCH` and `UPDATE` statements.
- Restructure the loop.

We examine the first two steps below. The last step is discussed in Section 6.

5.2.2. *Generation of SQL cursor declarations*

5.2.2.1. Syntax. A VSAM processing loop can have multiple conditions controlling its termination. It is necessary to take into account all these conditions in the search condition of a `DECLARE CURSOR`, because an inaccurate search condition in an SQL query leads to significant performance degradation, as illustrated in Figure 1. The format of an SQL `DECLARE CURSOR` is:

```
DECLARE <cursor name> CURSOR FOR
SELECT <attribute list> FROM <table name>
WHERE <search conditions> [ORDER BY clause]
```

We can automatically generate the cursor name, and use an asterisk '*' for the attribute list to represent all the attributes of the table, because VSAM operations always manipulate records as a whole. The `<table name>` is the counterpart of the `FD` name in the `SELECT` statement for the data set. Since VSAM sequential processing typically accesses records in ascending order, search conditions can be formed by identifying the conditions that control the start and end points of the processing sequence. We use flow analysis to identify the sequential processing loops. This involves three components:

- Identification of sequential processing loops, i.e., the loop that contains sequential VSAM operations.
- Identification of loop entry conditions, i.e., the conditions that select the starting record to be processed by the loop.
- Identification of loop exit conditions, i.e., the conditions that terminate the processing loop.

5.2.2.2. Identification of sequential processing loops. Identification of loops in programs can be carried out by flow analysis, and is well understood by the optimizing compiler community (Hecht, 1977; Kuck *et al.*, 1981; Aho, Sethi and Ullman, 1986; Ferrante, Ottenstein and Warren, 1987). We built a COBOL parser and dependence analyzer to perform flow analysis on COBOL programs with VSAM operations. We then used dominator information (Aho, Sethi and Ullman, 1986, p. 602) to find loops in the flow graph generated by the dependence analyzer.

Most structured COBOL programs implement loops by using `PERFORM` statements. These loops have only a single entry point. To determine if a single-entry loop is a VSAM sequential processing loop, we check all the paths from the entry point to the tail of the back edge (i.e., the edge whose head is the entry point in the flow graph). If any of these paths contains VSAM sequential operations, the loop is a sequential processing loop.

Some early COBOL programs use `GO TO` statements to form loops. In the example shown in Figure 1, two `GO TO` statements are used: one to go to the top of the loop as long as the loop continues, and the other to exit the loop when a termination condition is met. A `GO TO`-based loop paragraph like this usually uses one or more `GO TO` statements to go to the beginning of the loop paragraph, to continue the loop when looping conditions remain true. It also uses one or more `GO TO` statements to go to the end of the loop paragraph when loop termination conditions become true. These `GO TO`-based loops are reducible, and automated restructuring tools (e.g., Structured Retrofit) are available to transform them into structured loops. However, some unstructured programs use `GO TO` statements to jump to different points inside a loop, or jump out of the loop to different locations, resulting in non-reducible flow graphs, and making it hard to restructure. We shall not discuss these non-reducible unstructured loops further in this paper.

5.2.2.3. Identification of entry condition. There are two ways to specify the entry of a sequential processing loop. One is by default the first record of the data set, and the other is the record located by a `START` statement. This second way is not available for ESDS data sets.

Example. Consider the following COBOL program segment. There is no `START` statement between the `OPEN` and `PERFORM` statements. Since the file pointer is always set to the beginning of the data set after it is opened, this means that the starting point of the processing sequence is the first record, i.e., the one with the lowest key in the sorted or indexed `OLD-MASTER-FILE`.

```
MAIN.
    OPEN INPUT OLD-MASTER-FILE.
    PERFORM PROCESS-OLD-MASTER-FILE
        UNTIL LOW-KEY = HIGH-VALUES.
    ...
PROCESS-OLD-MASTER-FILE.
    READ OLD-MASTER-FILE
    AT END
        MOVE HIGH-VALUES TO LOW-KEY.
```

Since the records in a data set are sorted or indexed, the entry condition in this example is equivalent to `KEY >= LOW-VALUE`, which can be omitted because the `LOW-VALUE` is the minimum value of the key.

Example. Consider the following COBOL program segment.

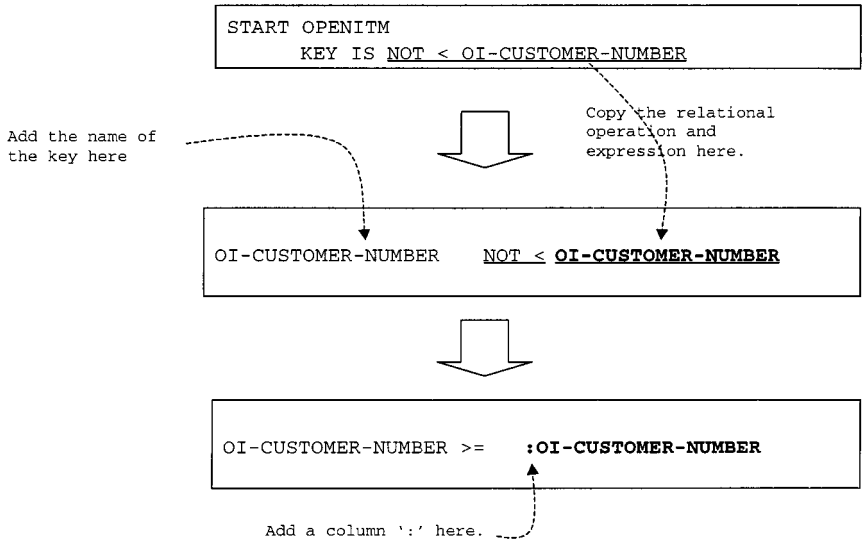

Figure 5. Example of converting a KEY *clause to an entry condition*

```
OPEN INPUT OPENITM.
MOVE 199 TO OI-CUSTOMER-NUMBER.
START OPENITM
    KEY IS NOT < OI-CUSTOMER-NUMBER
END-START.
PERFORM 300-PREPARE-OPEN-ITEM-LINES
    UNTIL OPENITM-EOF.
CLOSE OPENITM.
```

After the OPEN statement, a START statement sets the starting point to the first record for which the key value is not less than the OI-CUSTOMER-NUMBER, provided the data set is sorted in ascending order of the key. The loop starts with this record and continues until the end of the file. In this example, we can directly obtain the entry condition by rewriting the condition in the KEY clause to OI-CUSTOMER-NUMBER $\geq$:OI-CUSTOMER-NUMBER, where the OI-CUSTOMER-NUMBER to the left of the relational operator is the name of the key of the data set, and the :OI-CUSTOMER-NUMBER to the right is a host variable.

In general, the format of a START statement is START <file name> <key clause>, and the format of a key clause is KEY [IS] <relational operator> <expression>. The key clause specifies the condition for choosing the starting record. We can generate the entry condition of the processing sequence by adding the name of the key in the front of the KEY clause. We can then prefix a column sign ':' to all the other variables used in the key clause to convert them to the host variables. Figure 5 illustrates these two steps.

In some complex or ill-formed programs, the starting point of a processing sequence is determined by the selective execution of one of the multiple START statements. In such a case, we should consider the conditions used to select the START statements, and an entry condition is

considered as the condition in the START statement, in conjunction with the condition that selects this START statement. However, this form of programming style can often be avoided by factoring out the START statements.

Example. Consider the following COBOL program segment:

```
OPEN INPUT OPENITM.
IF OI-MIN-CUSTOMER-NUMBER < 199
    MOVE 199 TO OI-CUSTOMER-NUMBER
    START OPENITM
        KEY IS NOT < OI-CUSTOMER-NUMBER
    END-START
ELSE
    MOVE 299 TO OI-CUSTOMER-NUMBER
    START OPENITM
        KEY IS NOT < OI-CUSTOMER-NUMBER
    END-START
END-IF.
PERFORM 300-PREPARE-OPEN-ITEM-LINES
      UNTIL OPENITM-EOF.
CLOSE OPENITM.
```

We can factor out the START statements from the two sides of the IF statement and make them share the same one, since they are clones.

5.2.2.4. Identification of exit conditions. Most structured COBOL programs use one of the following three methods to terminate a sequential processing loop:

- Evaluate the condition in the UNTIL clause, and exit if true.
- Evaluate the conditions in the IF statements inside the loop, and execute a GO TO statement to exit the loop if true (or false).
- Evaluate the condition in an IF statement inside the loop and set a flag when the condition is true (or false). Setting the flag to true or false then causes the condition in the UNTIL clause to be evaluated as true, and terminates the loop.

Not all these conditions can be included in the SQL query expression. They should satisfy the following two additional tests:

- The condition must use at least one field of the data set. This is necessary because search conditions in an SQL query are evaluated on all the rows of a table. If the condition does not contain any table attribute, it is then irrelevant to the search.
- The host variables (i.e., the variables other than data set fields) in the condition can be evaluated before entering the loop—i.e., they can be loop-independent. This is because all the qualified rows are retrieved by the OPEN cursor statement before entering the sequential processing loop. If any of the host variables in the condition are loop-dependent, the condition cannot be incorporated into the search condition.

We can use dependence analysis to identify all three types of exit condition. Let *L* be a loop that processes a sequence of records in data set *D*. Suppose *UC* is the condition in the UNTIL clause of the loop *L*. We check if the condition *UC*

- uses any field of the data set *D*, and
- does not have any host variable, whose value is loop-dependent.

If the answer to both is YES, the condition *UC* can be used as a part of the search condition. Otherwise, we use dependence analysis to

- select all the reaching definitions that are enclosed in the loop *L* and used in the condition *UC*, and
- find the control predicates that are within the scope of the loop, and control these reaching definitions.

We then check each of these control predicates against the two tests. If a predicate passes the tests, we can use it as a part of the search condition too.

Finally, we find all the GO TO statements enclosed in the loop *L*. For each GO TO that jumps out of the loop, find the control predicates that are within the loop scope and control the GO TO statement. Check each of these control predicates against the same two tests. Choose it as a part of the search condition if it passes the tests. To illustrate, we present several examples of the identification of exit conditions from a loop.

Example. Consider the following program segment. The condition COUNT > 100 cannot be used as a part of the search condition, because the value of the variable COUNT is dependent on each iteration of the loop, and hence cannot be evaluated before entering the loop.

```
PERFORM
    UNTIL COUNT > 100
    ...
    ADD +1 TO COUNT
END-PERFORM.
```

Example. Consider the following program segment. Suppose MY-KEY is the key field of MY-DATA-SET. The condition MY-KEY > 100 should be used as a part of the search condition of a cursor declaration because it satisfies the two tests.

```
OPEN MY-DATA-SET.
PERFORM
    UNTIL MY-KEY > 100
    READ MY-DATA-SET
END-PERFORM.
```

Example. Consider the following program segment. Suppose MY-KEY is a field of MY-DATA-SET. The condition EXIT-STATUS = 'Y' in the UNTIL clause does not pass the first test, i.e., it contains no fields of MY-DATA-SET, but the condition MY-KEY > 100 in the IF statement enclosed in the loop sets EXIT-STATUS to 'Y' when it is evaluated as true. Therefore, the condition MY-KEY > 100 should be used as a part of the search condition.

Table 2. An example of exit condition identification

COBOL with VSAM	COBOL with SQL
MOVE 100 TO A1. START AFILE KEY IS = A1. PERFORM READ AFILE ... **IF A1 > 200** **GO TO END-P** UNTIL FALSE. END-P.	DECLARE C1 CURSOR FOR SELECT * FROM A WHERE **(A1 >=:A1) AND (A1 <=200)** ORDER BY A1 ... MOVE 100 TO A1. OPEN C1. PERFORM FETCH C1 INTO :A1, :A2, ... UNTIL **SQLCODE = '0'.**

```
OPEN MY-DATA-SET.
...
MOVE 'N' TO EXIT-STATUS.
PERFORM
    UNTIL EXIT-STATUS = 'Y'
    ...
    READ MY-DATA-SET
    IF MY-KEY > 100
        MOVE 'Y' TO EXIT-STATUS
    END-IF
END-PERFORM.
```

Example. Consider the example in Figure 1 again. The condition `A1 > 200` in the `IF` statement controls the `GO TO` statement that exits the loop. Since `A1` is the key of the data set `AFILE`, the condition `A1 > 200` passes the two tests and can be used as a part of the search condition of the cursor declaration. Its counterpart in the cursor declaration is `A1 ≤ 200`. The converted code accesses exactly the same records as the original code, as seen in Table 2.

5.2.2.5. Combination of entry/exit conditions. The last example in the previous section also shows that we cannot simply put the identified entry and exit conditions together to form the search condition of the cursor declaration. We need to perform some transformation on them. Unfortunately, we cannot fully automate this transformation process. In this section, we discuss some heuristics and guidelines on how to combine these entry and exit conditions into a search condition of a cursor declaration statement.

First, process the syntax of each individual entry and exit condition. The syntax of relational operators used in COBOL VSAM statements may be different from those in embedded SQL. These relational operators can be automatically translated into the desired ones in embedded SQL. Moreover, the entry/exit condition may use non-attribute variables that are none of the fields of the

data set. These non-attribute variables are considered host variables in embedded SQL, and hence, they should be prefixed with a column ':', the way to denote a host variable in a SQL statement.

Second, perform a logical transformation on each entry and exit condition. Since the loop processes the records in ascending (or chronological) order, the entry and exit conditions specify the first and last in the processing sequence, respectively. Therefore, if the relational operator in an entry condition is an equal '=', the operator should be transformed to greater-than-or-equal-to '≥'. The logical transformation on an exit condition is not so straightforward and needs human intervention.

Example. Consider the following program segment, and assume that the records are sorted in ascending order by the key field. Suppose `MY-FIELD-A` and `MY-FIELD-B` are a field of `MY-DATA-SET`, and `MY-KEY` is the key field.

```
OPEN MY-DATA-SET.
PERFORM
    UNTIL MY-FIELD-A > MY-FIELD-B
    ...
    READ MY-DATA-SET
END-PERFORM.
```

The condition `MY-FIELD-A` > `MY-FIELD-B` in the `UNTIL` clause passes the two tests, but cannot be used in the search condition with a negation or other simple transformation on the relational operator. The reason is that the VSAM loop assumes that the records are sorted in ascending order of the key field `MY-KEY` in the data set. Hence, the condition `MY-FIELD-A` > `MY-FIELD-B` terminates the loop when it reaches the first record in the sequence that satisfies the condition. However, the same assumption does not hold for the SQL tables. If we simply negate the condition `MY-FIELD-A` > `MY-FIELD-B`, and use the search condition of the cursor declaration, the cursor declaration will retrieve all the rows in the table that satisfy the condition `NOT(MY-FIELD-A` > `MY-FIELD-B)`, which is wrong.

We need to take special care of this type of exit condition as neither side of a relational operator uses the key fields of the data set. Fortunately, most of the exit conditions in the industrial programs we encountered do use the key fields. For such an exit condition, if a relational operator is an equal '=', the operator should be changed to less-than-or-equal-to '≤', or a greater-than-or-equal-to '≥', depending on how the key field is used in the condition; otherwise, we can simply negate the condition.

Finally, we determine how to connect the transformed entry and exit conditions together to form a complete search condition. This is rather complex. We use the following three steps:

- Connect all the entry conditions using logical AND or OR connectives.
- Connect all the exit conditions using logical AND or OR connectives.
- Connect the composite entry condition and the composite exit condition with a logical AND connective.

Algebraic operations can be performed to simplify the resulting search conditions. The following are some heuristic rules we use to connect two exit conditions.

Heuristic rule 1. Two exit conditions can be connected using the logical OR connective if they appear in two disjoint `IF` statements enclosed in the loop.

Example. Consider the following segment. `COND1` and `COND2` are used in two disjoint `IF` statements, respectively. We can connect them using the logical OR connective. The first example uses an implicit logical OR.

```
LOOP.
    ...
    IF COND1 THEN
        GO TO END-OF-LOOP
    END-IF
    IF COND2 THEN
        GO TO END-OF-LOOP
    END-IF
    ...
END-OF-LOOP.
```

The second example uses an explicit logical OR to reduce the number of lines of code.

```
LOOP.
    ...
    IF COND1 OR COND2 THEN
        GO TO END-OF-LOOP
    END-IF
    ...
END-OF-LOOP.
```

Heuristic rule 2. Two exit conditions can be connected with the logical AND connective if one controls the other.

Example. Consider the following segment. `COND1` and `COND2` are used in two nested `IF` statements. `COND1` controls `COND2.` We connect them using the logical AND connective. The first example uses an implicit logical AND.

```
LOOP.
    ...
    IF COND1 THEN
        IF COND2 THEN
            GO TO END-OF-LOOP
        END-IF
    END-IF
    ...
END-OF-LOOP.
```

The second example uses an explicit logical AND to reduce the appearance of nesting in the `IF` statements.

```
LOOP.
    ...
    IF COND1 AND COND2 THEN
```


Table 3. Conversion between sequential I/O statements and SQL statements

COBOL/VSAM IO statements	SQL manipulative statements
READ [NEXT]	FETCH
REWRITE	UPDATE CURRENT
WRITE	INSERT

```
        GO TO END-OF-LOOP
    END-IF
    ...
END-OF-LOOP.
```

5.2.3. *Converting sequential I/O statements to SQL statements*

Sequential VSAM I/O operations are used to read and update the existing records, or append (write) a new record to the end of the data set. Other VSAM I/O operations can only be applied to random access mode or KSDS and RRDS data sets. After generating the cursor declaration for the sequential loop, we can convert the sequential VSAM I/O statements to SQL manipulative statements in a straightforward way (see Table 3).

Each sequential READ statement is logically equivalent to an SQL FETCH statement plus an optional exception handling IF statement. The syntax of a sequential READ statement is:

```
READ <file name> [exception handling]
```

We can replace the exception handling clause and the associated COBOL statements with a COBOL IF statement like this:

```
IF SQLCODE = exception code THEN
<exception handling COBOL statements>
```

The variable SQLCODE is a special status variable used by the embedded SQL. Whenever an SQL statement is executed, the value of SQLCODE is updated to indicate the status of the operation.

Example. Consider a COBOL VSAM `READ` statement:

```
READ KAMOKU-MST
    AT END
        MOVE 9 TO FLAG-2-ACCOUNT
        GO TO KANJYO-READ-EXIT
END-READ.
```

The above `READ` statement can be converted to an SQL `FETCH` statement followed by a COBOL `IF` statement:

```
EXEC SQL FETCH DB2_CUR_0 INTO
    : H-DB2-MASTER-TYPE-CODE
```



```
    INDICATOR :F-DB2-MASTER-TYPE-CODE,
    {other host variables and indicators}
END-EXEC
IF SQLSTATE NOT = '00000'
    MOVE 9 TO FLAG-2-ACCOUNT
    GO TO KANJYO-READ-EXIT
END-IF.
```

Note that DB2_CUR_0 is a cursor name declared in a cursor declaration that retrieves the rows from the table corresponding to the data set KAMOKU-MST. The host variables are used to carry the values out of the columns in the SQL table.

A sequential REWRITE statement updates the record currently pointed to by the file pointer. We can convert it to an SQL UPDATE statement.

Example. Consider the following COBOL VSAM code segment:

```
MOVE INVENTORY-MASTER-RECORD TO MASTER-RECORD-AREA.
REWRITE INVMAST.
```

This code segment can be converted into the following code segment with an SQL UPDATE statement:

```
MOVE INVENTORY-MASTER-RECORD TO MASTER-RECORD-AREA.
MOVE MR-ITEM-NO TO H-DB1-MR-ITEM-NO.
MOVE MR-ITEM-DESC TO H-DB1-MR-ITEM-DESC.
...
MOVE MR-ON-ORDER TO H-DB1-MR-ON-ORDER.
EXEC SQL UPDATE INVMAST SET
    MR_ITEM_NO = :H-DB1-MR-ITEM-NO
    MR_ITEM_DESC = :H-DB1-MR-ITEM-DESC
    ...
    MR_ON_ORDER = :H-DB1-MR-UNIT-PRICE
    WHERE CURRENT OF DB1_CUR_0
END-EXEC
IF SQLSTATE NOT = '00000'
    EXEC SQL ROLLBACK WORKD END-EXEC
ELSE
    EXEC SQL COMMIT WORK END-EXEC
END-IF.
```

In the above code segment, variables prefixed with MR- are fields of the record MASTER-RECORD-AREA, and variables prefixed with H-DB1- are the corresponding host variables that are used to exchange data between the MASTER-RECORD-AREA and the SQL table. The WHERE clause in the SQL UPDATE statement indicates the operation to be performed on the row at the current position of the cursor DB1_CUR_0. The IF statement following the UPDATE statement checks if the operation is successful. If it is unsuccessful, an SQL ROLLBACK statement is executed to restore the table to its previous state; otherwise, an SQL COMMIT statement is executed to make the update permanent.

Table 4. Mapping random VSAM operation to SQL statement

Random VSAM operation	Embedded SQL statement

`READ <FD name>`	`SELECT *`
	`INTO <host variable list>`
	`FROM <table name>`
	`WHERE <key> = <:record key variable>`

`WRITE <FD name>`	`INSERT INTO <table name> <column list>` `VALUES (host variable list)`
`DELETE <FD name>`	`DELETE FROM <table name>` `WHERE <key> = <:record key variable>`

`REWRITE <FD name>`	`UPDATE <table name>`
	`SET {column = <scaler expression>}`
	`WHERE <key> = <:record key variable>`

5.3. Random and dynamic access methods

Compared to the sequential access mode, the conversion for the random access mode is straightforward. A random VSAM operation accesses only one record at a time. The record accessed is determined by the key value stored in the record key variable. To perform a random access, the key value of the record sought is assigned to the record key variable. Then, an I/O operation is issued to access the record with that key value. If the key field of the data set is a scalar, an equivalent SQL search condition can be generated in the form of

```
<key field> = <:record key variable>
```

where the `<key field>` is an attribute of the SQL table, and `<:record key variable>` is a host variable holding the value of the key of the row to be accessed. If the record key consists of multiple fields, the record accessed is determined by the values of all the individual fields of the key. Since the host variables for embedded SQL must be scalar variables, two host variables are introduced for the composite key, one for each field. The single `MOVE` statement that assigns a value to the composite key is also replaced with two `MOVE` statements that assign values to individual fields of the key. Finally, a random VSAM operation is translated into an SQL statement using Table 4.

Dynamic processing is a mixture of sequential and random processing. With the dynamic access mode, records can be read sequentially or randomly, depending on the type of `READ` operation. Consecutive sequential operations form a *sequential access block* (SAB), and consecutive random operations form a *random access block* (RAB). Therefore, the dynamic processing can be viewed as an alternating sequence of SABs/RABs. The dynamic access mode is rarely used, and not recommended because it is hard to understand and maintain. Also, any dynamic processing can usually be replaced with sequential or random processing (the proof of this claim is beyond the scope of this paper). We found that it is hard to convert the dynamic access mode. So far, we do not have an efficient and effective method to convert the dynamic access mode automatically. One of the main issues is how to separate out SABs and RABs if we want to treat them differently. Since a random `READ` statement reads a record based on the value of the record key field, it is independent

of other read operations. Two random READ statements are in the same RAB if and only if there is no sequential READ operation in between. However, sequential read operations often group together and span many statements in processing a sequence of records. It is hard to locate the start and end points of each SAB. For example, there may be two paths originating from a program point after a random VSAM operation. One of the two paths contains an RAB/SAB sequence, while the other contains a different RAB/SAB sequence. Furthermore, these two sequences may be enclosed in a loop and entwine.

6. RESTRUCTURING LOOPS

The main task of this step is to remove the redundant IF statements from the enclosing loop and replace the exit condition in the UNTIL clause. An enclosed redundant IF statement is one that implements an exit condition, where neither of its two sides contains statements other than the loop exit statement (e.g., a GO TO statement) or definitions of control flags. The original condition in the UNTIL clause is always replaced with SQLCODE = '0', which indicates the end position in the retrieved result.

Example. In Table 2, since there is only one GO TO statement inside the IF statement, it becomes redundant in the converted code with SQL statements and is removed. The original condition in the UNTIL clause is replaced with SQLCODE = '0', which indicates the end of the processing.

7. IMPLEMENTATION

We have implemented a prototype tool that can automatically convert sequential and random VSAM operations in COBOL programs to embedded SQL statements. It implements the approaches discussed in the previous sections. Figure 6 shows the block diagram of the tool.

All the blocks request information from the data dictionary and the SCDG (statement-level COBOL dependence graph) (Huang, Tsai and Subramanian, 1996). The data set identification and VSAM I/O operation identification create a table to store the data set properties, record structure, and pointers to the I/O statements. Each data set has an entry in the table (see Figure 7). If the processing on a data set is sequential, we identify the sequential processing loops by checking all the loops in the SCDG that contain sequential READ operations on that data set. Then, based on the information collected in the data set table and data dictionary, the relational schema, the host variable declaration, and the SQL statements are generated. Finally, the VSAM data set definitions are replaced with the relational schema and host variable declarations, and the VSAM operations are replaced with SQL statements. Note that programmers' review of the tool results is needed to correct any error conditions and make any changes necessary.

8. VERIFICATION AND VALIDATION

The converted database schemas, databases and application programs should be verified and validated. Before regression testing the migrated software, code review or inspection should be performed to identify and mitigate conversion errors. To facilitate the code review or inspection, we use the following conversion conventions:

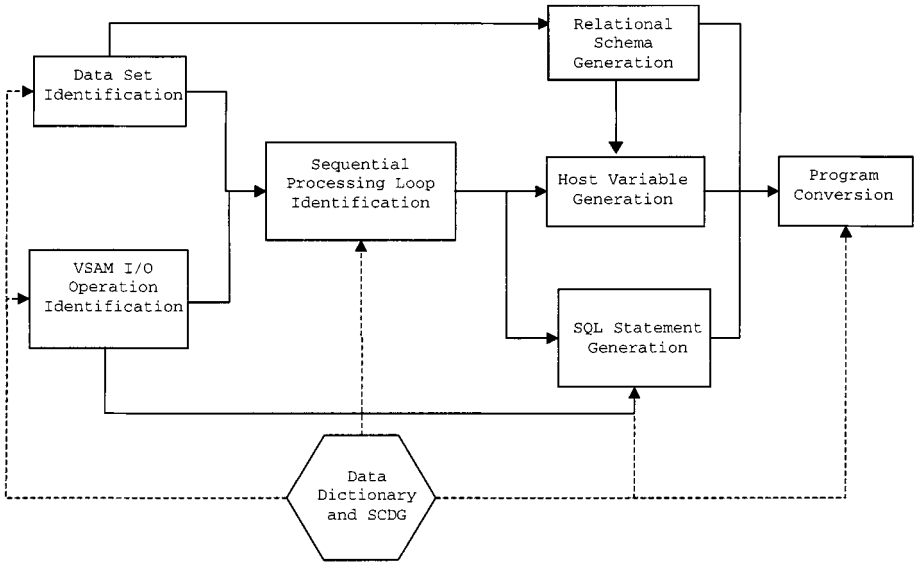

Figure 6. Block diagram of the VSAM re-engineering tool

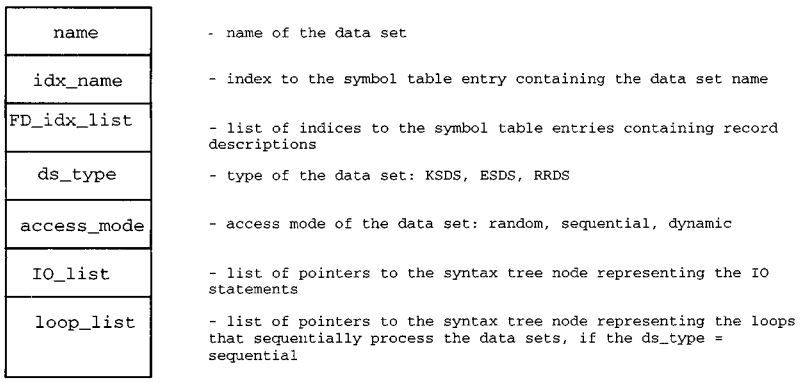

Figure 7. Example of a layout of an entry in the data set table

- Comment out, rather than remove, the VSAM components from the original source code.
- Use `diff` (a Unix utility) to generate a side-by-side listing with the old code in the first column and the converted code in the second column, to facilitate a side-by-side comparision. Table 5 is an example of such a listing.

In addition to existing regression testing suites, new test cases should be developed to validate and verify the correctness of the modified control and data flows. The regression testing suite should be run on both the migrated software and the original software to verify congruent results.

Finally, the performance of the migrated software should be benchmarked. Manual performance tuning may be required.

Table 5. An example of sdiff between the old and converted code

COBOL+VSAM		COBOL+SQL
`MOVE W-KEY TO T-KEY.`	*	`MOVE W-KEY TO T-KEY.`
		`MOVE W-KEY TO :H-KEY.`
`START T2-FILE`	*	`START T2-FILE`
`KEY IS = T-R-KEY.`		`KEY IS = T-R-KEY.`
		`EXEC SQL DECLARE`
		`T-CUR-001 CURSOR FOR ~`
		`WHERE T_KEY = :H-T-KEY`
		`END-EXEC`
		`EXEC SQL FETCH ~ END-EXEC`
`IF T-ST NOT = '00'`	*	`IF T-ST NOT = '00'`
`GO TO ERR-PROC`	*	`GO TO ERR-PROC`
`END-IF.`	*	`END-IF`
		`IF SQLSTATE NOT = '00000'`
		`EXEC SQL`
		`CLOSE T-CUR-001`
		`END-EXEC`
		`GO TO ERR-PROC`
		`END-IF.`

9. CONCLUSION

Converting VSAM operations in COBOL programs to embedded SQL is a way of migrating legacy database applications to relational database and client–server applications. Manual conversion without tool assistance is possible but labor intensive. Since it is difficult for programmers to analyze dependencies between conditions that control the execution of sequential VSAM processing loops, programmers may not be able to identify all possible loop termination conditions, and often produce SQL queries that are less efficient than the original VSAM operations. This paper proposes dependence analysis based techniques to assist programmers in identifying VSAM operations and generating efficient SQL queries. The proposed approach and techniques, as well as a tool that implements them, have been tried on many COBOL programs from industry. The appendix gives a detailed example.

References

Aho A, Sethi R, Ullman J. 1986. *Compilers: Principles, Techniques, and Tools*; Addison-Wesley: Reading MA; 796 pp.

Brumbaugh L. 1993. *VSAM: Architecture, Theory and Applications*; McGraw-Hill: New York NY; 396 pp.

Chen X, Tsai W, Joiner J, Gandamaneni H, Sun J. 1994. Automatic variable classification for COBOL programs. *Proceedings COMPSAC 94*; IEEE Computer Society Press: Los Alamitos CA; pp. 432–437.

Ferrante J, Ottenstein K, Warren J. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* **9**(3):319–349.

Gray R, Bickmore T, Williams S. 1995. Reengineering COBOL systems to ADA, CD-ROM *Proceedings of the Seventh Annual Air Force/Army/Navy Software Technology Conference*; Utah State University: Logan UT; 11 pp.

Hecht M. 1977. *Flow Analysis of Computer Programs*; North Holland Publishing Co.: New York NY; 232 pp.

Huang H, Tsai W, Subramanian S. 1996. Generalized program slicing for software maintenance. *Proceedings of the Eighth International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*; Knowledge Systems Institute: Skokie IL; pp. 261–268.

Huang H, Tsai W, Bhattacharya S, Chen X, Wang Y, Sun J. 1998. Business rule extraction techniques for COBOL programs. *Journal of Software Maintenance: Research and Practice* **10**(1):3–35.

Kuck D, Kuhn R, Padua D, Leasure B, Wolfe M. 1981. Dependence graphs and compiler optimizations. *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*; ACM Press: New York NY; pp. 207–218.

Ong C, Tsai W. 1993. Class and object extraction from imperative code. *Journal of Object-Oriented Programming* **6**(2):58–60, 68.

Polak W, Nelson L, Bickmore T. 1995. Reengineering IMS databases to relational systems. CD-ROM *Proceedings of The Seventh Annual Air Force/Army/Navy Software Technology Conference*; Utah State University: Logan UT; 9 pp.

Rugaber S, Doddapaneni S. 1993. The transition of application programs from COBOL to a fourth generation language. *Proceedings Conference on Software Maintenance*; IEEE Computer Society Press: Los Alamitos CA; pp. 61–70.

Ulman J. 1988. *Principles of Database and Knowledgebase Systems*, Volume 1; Computer Science Press: Rockville MD; p. 77

Authors' biographies:

Hai Huang is a senior software engineer at Guidant Corporation. He earned his B.S. and M.S. in Computer Science from the University of Science and Technology of China, and his M.S. and Ph.D. in Computer Science from the University of Minnesota, Minneapolis Minnesota. His current interest is in safety-critical software verification, automated real-time system testing, and software engineering. His email address is: hai.huang@guidant.com

Wei-Tek Tsai is a Professor of Computer Science at the University of Minnesota, Minneapolis Minnesota. He earned his S.B. in Computer Science and Engineering from MIT at Cambridge, MA, and his M.S. and Ph.D. in Computer Science from the University of California at Berkeley. His current interest is in Internet computing and software engineering. He is on the editorial board of the IEEE Computer Society Press, and was the Program Chair of the IEEE International Conference on Computer Software and Applications in 1997. His email address is: tsai@cs.umn.edu

APPENDIX. AN EXAMPLE OF VSAM DATABASE RE-ENGINEERING

In this appendix, we present an example of converting a sequential processing VSAM program to a program with embedded SQL statements. The result of the conversion was produced automatically

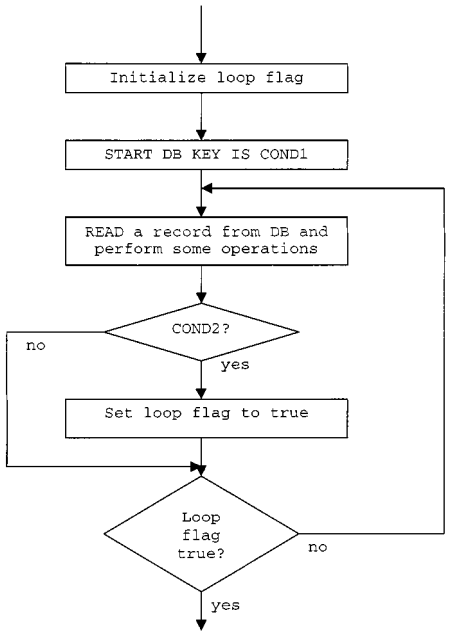

Figure 8. Outline of the control structure of the example VSAM program

by the tool discussed in Section 7. Figure 8 illustrates the outline of the control structure for the example.

The tool uses the following steps to perform the conversion:

- Locate the `START` and/or `OPEN` statement.
- Find the outermost loop that encloses any read operations on the database, which is opened or started by the `OPEN` or `START` statement.
- Get the exit condition from the `UNTIL` clause.
- Find all the assignment statements that affect the value of the exit condition of the loop.
- Find all the conditions that control the execution of the above assignment statements.

All the above control conditions are potential indirect loop exit conditions, which can be used in SQL cursor declarations to reduce the number of rows to be retrieved. In this example VSAM program:

- The sequential processing loop starts at line 00440, and the variable `FLAG-2-ACCOUNT` is used as a flag controlling the exit of the loop.
- The `START` statement at line 004150 in the left column specifies a start condition

  ```
  KEY IS NOT < N-NAME-MASTER-KEY-1
  ```

  that is used in the SQL cursor declaration, starting at line 004240 in the right column, as a part of the search condition.

- The `IF` statement at line 005190 in the right column controls the value of `FLAG-2-ACCOUNT`, and hence, the condition of the `IF` statement is used as a part of the search conditions of the cursor declaration statement.

Table 6 lists the result of the example. The first column lists the original code, and the second column lists the converted code produced by the tool.

Table 6. A VSAM to SQL conversion example

COBOL with VSAM

```
004020 USER-INIT-ROUTINE SECTION.
004030     OPEN  INPUT  KAMOKO-MST.
004040
004050     MOVE  0  TO  FLAG-2-ACCOUNT.
004060
004080     SET  IX  TO 0.
004090     MOVE ALL '9' TO TABLE-SUMMARY.
004100
004110     MOVE  SPACE  TO  N-NAME-MASTER-KEY-1.
004120     MOVE  9      TO  N-MASTER-TYPE-CODE.
004130     MOVE  0      TO  N-ACCOUNT-KIND-CODE.
004140
004150     START   KAMOKU-MST USING KEY IS
004155          NOT < N-NAME-MASTER-KEY-1
004160         INVALID KEY
004170         DISPLAY '%%% KNA2 INVALID'
004175                 N-NAME-MASTER-KEY-1  UPON  SYSOUT
004180         MOVE  50  TO  RETURN-CODE
004190         STOP  RUN
004200     END-START.
004410
```

COBOL with SQL

```
004020 USER-INIT-ROUTINE SECTION.
004030*     OPEN  INPUT  KAMOKU-MST.
004040
004050     MOVE  0  TO  FLAG-2-ACCOUNT.
004070
004080     SET  IX  TO 0.
004090     MOVE ALL '9'  TO  TABLE-SUMMARY.
004100
004110     MOVE  SPACE  TO  N-NAME-MASTER-KEY-1.
004120     MOVE  9      TO  N-NASTER-TYPE-CODE.
004130     MOVE  0      TO  N-ACCOUNT-KIND-CODE.
004140
004150*     START   KAMOKU-MST USING KEY IS
004155*          NOT < N-NAME-MASTER-KEY-1
004160*         INVALID KEY
004170          DISPLAY '%%% KNA2 INVALID'
004175                N-NAME-MASTER-KEY-1  UPON  SYSOUT
004180          MOVE  50  TO  RETURN-CODE
004190          STOP  RUN
004200      END-START.
004210*
004215     MOVE N-NAME-MASTER-KEY-1 TO
004218                 H-DB2-N-NAME-MASTER-KEY-1.
004220     MOVE N-MASTER-TYPE-CODE TO
004225                 H-DB2-N-NAME-MASTER-TYPE-CODE.
004230*
004240     EXEC SQL DECLARE DB_2CUR_0 CURSOR FOR
004250       SELECT    MASTER_TYPE_CODE,
004260          ACCOUNT_TYPE_CODE,
004270          ACCOUNT_TYPE_NAME,
004280          ACCOUNT_TYPE_NAME_N,
004290          SUMMARY_ACCOUNT_TYPE_CODE,
004300          CD_TYPE,
004310          ACCOUNT_LEVEL1,
004320          ACCOUNT_LEVEL8,
004330          ACCOUNT_LEVEL9,
004340          REGIST_DATE_1
004350       FROM KAMOKU_MST
004360       WHERE ( NAME_MASTER_KEY_1 >=
004365                :H-DB2-N-NAME-MASTER-KEY-1 ) AND
004370                NOT ( MASTER_TYPE_CODE <>
004375                            :H-DB2-N-MASTER-TYPE-CODE )
004380     END-EXEC
004390     EXEC SQL OPEN DB2_CUR_0  END-EXEC
004400*
004410
```


Table 6. Continued.

COBOL with VSAM	COBOL with SQL
004420 PERFORM KANJYO-READ-RTN.	004420 PERFORM KANJYO-READ-RTN.
004430 SET IX TO 0.	004430 SET IX TO 0.
004440 PERFORM KANJYO-SET-RTN	004440 PERFORM KANJYO-SET-RTN
004445 WITH TEST BEFORE	004445 WITH TEST BEFORE
004450 UNTIL FLAG-2-ACCOUNT = 9.	004450 UNTIL FLAG-2-ACCOUNT = 9.
004460	004460
004790 CLOSE KAMOKU-MST.	004790* CLOSE KAMOKU-MST.
004810 USER-INIT-ROUTINE-EXIT.	004810 USER-INIT-ROUTINE-EXIT.
004820 EXIT	004820* EXIT
004830	004830
004840 KANJYO-READ-RTN SECTION.	004840 KANJYO-READ-RTN SECTION.
004850 READ KAMOKU-MST	004850* READ KAMOKU-MST
004860 AT END	004860* AT END
004870 MOVE 9 TO FLAT-2-ACCOUNT	004870* MOVE 9 TO FLAT-2-ACCOUNT
004880 GO TO KANJYO-READ-EXIT	004880* GO TO KANJYO-READ-EXIT
004890 END-READ.	004890* END-READ.
005180	004900
	004910 EXEC SQL FETCH DB2_CUR_0 INTO
	004920 :H-DB2-MASTER-TYPE-CODE
	004925 INDICATOR :F-DB2-MASTER-TYPE-CODE,
	004930 :H-DB2-ACCOUNT-TYPE-CODE
	004935 INDICATOR :F-DB2-ACCOUNT-TYPE-CODE,
	004940 :H-DB2-ACCOUNT-TYPE-NAME
	004945 INDICATOR :F-DB2-ACCOUNT-TYPE-NAME,
	004950 :H-DB2-ACCOUNT-TYPE-NAME-N
	004955 INDICATOR :F-DB2-ACCOUNT-TYPE-NAME-N,
	004960 :H-DB2-SUMMARY-ACCOUNT-TYPE-CODE
	004965 INDICATOR :F-DB2-SUMMARY-ACCOUNT-TYPE-CODE,
	004970 :H-DB2-CD-TYPE
	004975 INDICATOR :F-DB2-CD-TYPE,
	004980 :H-DB2-ACCOUNT-LEVEL1
	004985 INDICATOR :F-DB2-ACCOUNT-LEVEL1,
	004990 :H-DB2-ACCOUNT-LEVEL8
	004995 INDICATOR :F-DB2-ACCOUNT-LEVEL8,
	005000 :H-DB2-ACCOUNT-LEVEL9
	005005 INDICATOR :F-DB2-ACCOUNT-LEVEL9,
	005010 :H-DB2-REGIST-DATE-1
	005015 INDICATOR :F-DB2-REGIST-DATE-1
	005020 END-EXEC
	005030 IF SQLSTATE NOT = '0000'
	005040 MOVE 9 TO FLAG-2-ACCOUNT
	005050 GO TO KANJYO-READ-EXIT
	005060 END-IF.
	005070 MOVE H-DB2-MASTER-TYPE-CODE TO
	005075 MASTER-TYPE-CODE.
	005080 MOVE H-DB2-ACCOUNT-TYPE-CODE TO
	005085 ACCOUNT-TYPE-CODE.
	005090 MOVE H-DB2-ACCOUNT-TYPE-NAME TO
	005095 ACCOUNT-TYPE-NAME.
	005100 MOVE H-DB2-ACCOUNT-TYPE-NAME-N TO
	005105 ACCOUNT-TYPE-NAME-N.
	005110 MOVE H-DB2-SUMMARY-ACCOUNT-TYPE-CODE TO
	005115 SUMMARY-ACCOUNT-TYPE-CODE.
	005120 MOVE H-DB2-CD-TYPE TO CD-TYPE.
	005130 MOVE H-DB2-ACCOUNT-LEVEL1 TO
	005135 ACCOUNT-LEVEL1.
	005140 MOVE H-DB2-ACCOUNT-LEVEL8 TO
	005145 ACCOUNT-LEVEL8.
	005150 MOVE H-DB2-ACCOUNT-LEVEL9 TO
	005155 ACCOUNT-LEVEL9.
	005160 MOVE H-DB2-REGIST-DATE-1 TO
	005165 REGIST-DATE-1.

Table 6. Continued.

COBOL with VSAM	COBOL with SQL
	005170*
	005180
005190 IF MASTER-TYPE-CODE OF	005190 IF MASTER-TYPE-CODE OF
005195 ACCOUNT-TYPE-NAME-MASTER-12	005195 ACCOUNT-TYPE-NAME-MASTER-12
005200 NOT = N-MASTER-TYPE-CODE	005200 NOT = N-MASTER-TYPE-CODE
005210 THEN	005210 THEN
005220 MOVE 9 TO FLAT-2-ACCOUNT	005220 MOVE 9 TO FLAT-2-ACCOUNT
005230 ELSE	005230 ELSE
005240* NEXT SENTENCE	005240* NEXT SENTENCE
005250 CONTINUE	005250 CONTINUE
005260 END-IF.	005260 END-IF.
005270 KANJYO-READ-EXIT.	005270 KANJYO-READ-EXIT.
005280* EXIT.	005280* EXIT.
005290	005290
005300 KANJYO-SET-RTN SECTION.	005300 KANJYO-SET-RTN SECTION.
005310 SET IX UP BY 1.	005310 SET IX UP BY 1.
005320	005320
005330 IF IX > 1500	005330 IF IX > 1500
005340 THEN	005340 THEN
005350 DISPLAY 'ABC 1500 TIMES' UPON SYSOUT	005350 DISPLAY 'ABC 1500 TIMES' UPON SYSOUT
005360 MOVE 50 TO RETURN-CODE	005360 MOVE 50 TO RETURN-CODE
005370 STOP RUN	005370 STOP RUN
005380 END-IF.	005380 END-IF.
005390	005390
005400 MOVE ACCOUNT-TYPE-CODE OF	005400 MOVE ACCOUNT-TYPE-CODE OF
005405 ACCOUNT-TYPE-NAME-MASTER-I2 TO	005405 ACCOUNT-TYPE-NAME-MASTER-I2 TO
005410 TAIL-ACCOUNT-KIND-CODE (IX).	005410 TAIL-ACCOUNT-KIND-CODE (IX).
005420 MOVE SUMMARY-ACCOUNT-TYPE-CODE OF	005420 MOVE SUMMARY-ACCOUNT-TYPE-CODE OF
005425 ACCOUNT-TYPE-NAME-MASTER-I2 TO	005425 ACCOUNT-TYPE-NAME-MASTER-I2 TO
005430 TAIL-SUMMARY-DEST-	005430 TAIL-SUMMARY-DEST-
005435- ACCOUNT-KIND-CODE (IX).	005435- ACCOUNT-KIND-CODE (IX).
005440 MOVE ACCOUNT-LEVEL8 OF	005440 MOVE ACCOUNT-LEVEL8 OF
005445 ACCOUNT-TYPE-NAME-MASTER-I2 TO	005445 ACCOUNT-TYPE-NAME-MASTER-I2 TO
005450 TAIL-ACCOUNT-LEVEL8 (IX).	005450 TAIL-ACCOUNT-LEVEL8 (IX).
005460 MOVE ACCOUNT-LEVEL9 OF	005460 MOVE ACCOUNT-LEVEL9 OF
005465 ACCOUNT-TYPE-NAME-MASTER-I2 TO	005465 ACCOUNT-TYPE-NAME-MASTER-I2 TO
005470 TAIL-ACCOUNT-LEVEL9 (IX).	005470 TAIL-ACCOUNT-LEVEL9 (IX).
005480 SET IX-MAX TO IX.	005480 SET IX-MAX TO IX.
005490	005490
005500 PERFORM KANJYO-READ-RTN.	005500 PERFORM KANJYO-READ-RTN.
005510 KANJYO-SET-RTN-EXIT.	005510 KANJYO-SET-RTN-EXIT.
005520* EXIT	005520* EXIT

